Processor Management in the Tera MTA Computer System

Authors:
Gail Alverson
Tera Computer Company
2815 Eastlake Avenue East
Seattle, Washington 98102
gail@tera.com
(206)325-0800
Simon Kahan
skahan@tera.com
Richard Korry
richk@tera.com

Keywords:
processor scheduling, swapping, gang scheduling, space-sharing, multithreading

Abstract

This paper describes the processor scheduling issues specific to the Tera MTA (Multi Threaded Architecture) computer system and presents solutions to classic scheduling problems. The Tera MTA exploits parallelism at all levels, from fine-grained instruction-level parallelism within a single processor to parallel programming across processors, to multiprogramming among several applications simultaneously. Consequently, processor scheduling occurs at many levels, and managing these levels poses unique and challenging scheduling concerns. We describe the processor scheduling algorithms of the user level runtime and operating system and describe the issues relevant to each. Many of the issues encountered and solutions proposed are novel, given the multithreaded, multiprogrammed nature of our architecture.

1. Introduction

The MTA is a multi-user system in which many parallel jobs of different sizes execute concurrently. Processors are multithreaded, with up to 16 jobs competing simultaneously for a processor's instruction streams. Parallel tasks therefore can space-share even within a single processor.

The Tera MTA supports two levels of processor scheduling (*Ousterhout, Zahorjan et al., Zahorjan et al., Gupta et al.*) The operating system scheduler manages processors at a gross level, and determines which jobs are executing at any moment. Once loaded, each job competes with other jobs for instruction streams without intervention from the operating system. The competition is guided by user-level runtime systems, one per job, that request and release streams in response to their respective jobs' parallel workloads. The runtime also works in concert with the compiler to schedule both automatically-generated and user-generated parallelism within a job.

This paper describes the processor scheduling issues specific to the Tera architecture. We outline the

scheduling policies of both the operating system and the user level runtime. First, in section 2, we familiarize the reader with the architecture of the Tera MTA. Section 3 describes the programming environment made available to the user. Section 4, on the user-level runtime environment, describes the scheduling done within user programs. The operating system is the subject of section 5, which describes the overall goals of processor scheduling among competing jobs. Section 6 describes the detailed issues and policies involved in processor scheduling. A summary and references are found at the end of the document as sections 7 and 8.

2. Architecture

The Tera MTA architecture, described in *Alverson et al.*, implements a physically shared-memory multiprocessor with multi-stream processors, interleaved memory units, and a packet-switched interconnection network. The memory is distributed and shared, and all memory units are addressable by every processor. The network can support a request and a response from each processor to a random memory location in each clock tick.

Each processor supports 16 protection domains and 128 instruction streams. A protection domain has a memory map and a set of registers that hold stream resource limits and accounting information. Consequently, each processor can be executing 16 distinct applications in parallel. Although fewer than 16 probably suffice to attain peak processor utilization, the surplus gives the operating system flexibility in mapping applications to processors. While two or three parallel jobs may sustain a high average parallelism executing on a processor, so may a mixture of parallel and sequential jobs.

A stream has its own register set and is hardware scheduled. On every tick of the clock, the processor logic selects a stream that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor, and by the network and memories as well, a new instruction from a different stream may be issued in each tick without interfering with its predecessors.

Provided there are enough instruction streams in the processor so that the average instruction latency is filled with instructions from other streams, the processor is fully utilized. Analogous to the provision of protection domains, the hardware provision of 128 streams per processor is more than enough to keep the processor busy. The extra streams let running jobs fully utilize the processor while other jobs are swapping in and out, and help saturate the processor during periods of higher memory latency (perhaps due to synchronization waits or memory contention).

Several architectural features are especially useful for scheduling. A *stream limit* counter in each protection domain lets the operating system impose a hardware-enforced bound on the streams that can be acquired by a job. Three user level instructions dynamically acquire and release streams in the protection domain of the invoking stream: `stream_reserve`, `stream_create`, and `stream_quit`. The novel `stream_reserve` instruction facilitates automatic program parallelization of programs; it grants the right to issue some or all of the requested number of `stream_create` instructions, each of which in turn will activate an idle stream and pass it a program counter and a data environment. Compiler generated scheduling code can compute a division of parallel work appropriate for the number of streams actually available, and do this before those streams begin consuming processor resources. The `stream_quit` instruction returns a stream to the idle state.

3. Programming Environment

Programs for the Tera are written in Tera C, C++ and Fortran. Parallelism can be expressed explicitly and implicitly. Explicit parallel programming is assisted through the use of *future variables* and *future statements*. A future variable describes a location that can eventually receive the result of executing a future statement, which is executed in parallel with its invoker.

The Tera MTA has full/empty bits on every word of memory which support an efficient implementation of futures. When a future statement starts, its future variable is set empty, causing subsequent reads to block. When the future statement completes, it writes its result to the future variable and sets it full, freeing those parallel activities that were waiting to read the variable. The user runtime is responsible for scheduling and executing future statements.

Implicit parallelism is identified by Tera's parallelizing compiler. Most of this is loop level parallelism: parallelism obtained by concurrently executing loop iterations. Other forms exploited include block level parallelism, executing independent blocks of code concurrently, and data pipelining (achieved only when explicit synchronization is provided).

The compiler schedules resources for much of the parallelism it generates. It generates instructions to allocate and start hardware streams, to schedule the work among the streams either statically or using a self-scheduling approach, and to release the streams. When parallelism is plentiful the compiler acquires streams on multiple processors. To do this it generates calls to the user runtime (which in turn may call the operating system) to create a stream in each remote protection domain. Compiler-generated code then acquires and releases streams in each domain based on the amount of parallelism.

In summary, the user level runtime schedules resources in response to explicit parallelism; compiler generated code adaptively schedules resources wherever implicit parallelism is detected.

4. Runtime Environment

The runtime is responsible for obtaining the best response time for the user's program while ensuring that

1. the program abides by rules imposed to ensure equitable access to processing resources, and
2. the program responds to anomalies (signals, traps) without endangering its own correctness.

Here, we describe features of the runtime pertaining only to the first constraint. That is, we describe the structure and policies of the runtime that support fast response to changes in parallelism without significantly reducing overall machine throughput.

Software Architecture

The software architecture of the runtime consists of a hierarchy of objects animating the processing behavior of the Tera. Many objects parallel those of the operating system.

A running user program is represented as a *task* consisting of a collection of *teams*, each of which occupies a protection domain. The protection domains are on distinct processors if this is possible. Practically speaking, it serves no purpose to put two teams of the same task on the same processor: the stream limit imposed by the operating system is large enough to enable a parallel team to more than

saturate the processor.

Each team has a set of *virtual processors* (vps), each of which is a process bound to a hardware stream. Note the distinction between a stream, a hardware entity, and a vp, a software entity. Every vp has a dedicated stream, but not every stream has an associated vp. Compiler generated parallelism makes use of streams without vps. Or a stream may simply be unused. A task starts with one team and one vp by default. The runtime is responsible for acquiring more vps if there is parallelism in the program and resources are available.

The set of active tasks at any moment is under the control of the operating system. The runtimes representing each of these tasks compete with one another for resources.

Scheduling Work

Work appears in two types of queues in the runtime. New futures are spawned into the *ready pool* of the task. The ready pool is a parallel FIFO based on `fetch&add`; see *Gottleib et al.* Futures that once were blocked but are now ready to run are placed in the *unblocked pool* of the team on which they blocked. Distributed unblocked pools are a convenient means to direct special work, such as a command to exit, to a particular team.

The runtime self-schedules work from the ready and unblocked pools. Vps from every team repeatedly select and run work from the pools. A vp's unblocked pool is given precedence because we would like to complete old work before starting new work. Executing work can result in allocation of additional instruction streams for compiler generated parallelism, creation of new work, and blocking of work. The vp executing work that creates new work or unblocks old work is responsible for placing that work on the ready or on the unblocked pool, respectively. If the vp blocks, it searches for other work on the unblocked and then the ready pool; that is, blocking does not lead to busy-waiting.

Some programs, especially those using divide-and-conquer algorithms, may do better with an ordering other than FIFO. A language extension `touch` (short for *touch-future*) assists with the effective execution of those programs (*Wagner and Calder* outline a more general version of touch called *leapfrogging*). When touch is applied to a future variable, the vp stops executing its current work and executes the future statement instead, unless the future is already executing. Thus the vp executes the work on which *its* work is waiting. This policy trades a few extra instructions if the future has already started for the overhead of blocking if it hasn't. Using touch makes future scheduling more LIFO than FIFO in character.

In practice, the user need not explicitly touch futures in divide-and-conquer programs. The compiler is able to infer divide-and-conquer recursive structure; when it discovers the storage class of a future variable is automatic, it generates code to perform the touch. The result is breadth-first scheduling of subproblems until vp creation lags behind subproblem creation; at that point the scheduling becomes depth-first. This dynamic schedule is both efficient regarding overheads and reactive to changes in the vp workforce.

Scheduling Stream Resources

The runtime is responsible for dynamically increasing and decreasing the number of vps it employs to execute the user program. This is important because the average size of the ready pool may vary

significantly over the lifetime of the program. Reasonably, a large ready pool merits more vps than a small ready pool, and vps acquired for periods of high parallelism should be retired during periods of low parallelism to improve efficiency.

The goals of the virtual processor strategy of the runtime mirror those of the runtime itself: an increase in the number of vps is attempted whenever it might reduce response time of the program, subject to the constraint that overall throughput of the machine should not be significantly reduced. Were acquisition and release of streams instantaneous, a policy of creating a new vp for each and every future would ensure minimum response time; retirement of vps whenever pools emptied would minimize impact on throughput. But there are overheads, and the runtime cannot foresee how pool sizes will change and how much computation any piece of work will demand. Nor does it have information regarding the behavior of the other programs with which it is competing. So there is an inherent tradeoff between minimizing response time, through zealous resource acquisition with delayed release, and maximizing throughput - assuming resources lag demand - through conservative acquisition with immediate release.

Virtual Processor Growth Policy

Because we wish to minimize the overhead involved when a vp adds work to the ready pool, the growth policy is implemented not by those vps but by a daemon mechanism that periodically assesses growth based on idleness and work available. The daemon creates twice the number of vps previously created whenever (1) there is *any* work to be done, and (2) no vps were idle since the last acquisition. For example, if there is initially one vp, after one period another will be created; after two periods, assuming both vps were kept busy, two more will be created; after three periods, supposing one of the four vps found nothing to do, even for a moment, no vps will be created; and after four periods, assuming all four vps were this time kept busy, one more vp will be created. If an acquisition request fails, in whole or in part, it is repeated after the next period has elapsed.

Our intention is to achieve a compromise in which the total overhead is at most a small constant fraction of the work intrinsic to the program, and the response time is similarly bounded from above by at most a small constant factor multiplied by the minimal response time, were all streams requested actually available. (For response time, an additive lower order term proportional to the logarithm of the maximum parallelism persists in our scheme, due to the delay of exponential growth versus instantaneous growth.) Such worst-case bounds should be achieved only when the parallelism is highly erratic in just the wrong way: we expect typical performance to be much closer to optimal.

Virtual Processor Retirement Policy

Just as important as the acquisition policy is the retirement policy: idle vps are released according to a schedule that inverts the acquisition schedule. That is, were a large number of vps to become idle at once, one would disappear after one period, two more after a second period, and so on unless work appeared to interrupt the process. This is necessary to avoid the anomalous behavior that would result were there to be brief sequential periods separating slightly longer but highly parallel periods at just the right frequency to cause repeated rapid growth and collapse. Were idle streams released after a constant period of time independent of their number, the constant factor in the response time bound becomes logarithmic in the maximum parallelism.

Because idle vps on the Tera consume as much processing power as vps performing memory-bound computations, our implementation never really allows more than one vp per team to be idle

concurrently: within a period, additional idle vps are killed, but remembered, and replaced the instant work appears. Creation and retirement within a team is through user level instructions, and so is fast enough to justify such a reactionary policy. Typically, this should make our upper bound for total work particularly pessimistic. A drawback is that streams, once released, may be acquired by competing teams and thus be unavailable when work reappears: when this happens, our bound for response time becomes arbitrarily optimistic. An alternative is to exploit the hitherto-unmentioned `stream_quit_preserve` instruction which causes a stream to stop executing but does not release it.

Team Growth and Retirement

The runtime uses unprivileged instructions to acquire more stream resources - up to a limit imposed by the operating system - on any team belonging to its task. Interaction with the operating system is required, however, when the runtime seeks to add new streams running in a different protection domain (on a different processor). This is because team creation requires resources other than streams: resources such as text memory and protection domains are globally controlled and initialized. The runtime interaction takes the form of an operating system call whose return value, like `stream_reserve`, must be checked for degree of success. Parameters to the call include the number of teams desired, a start-up function for each team's first vp (the operating system creates only the first stream on each team), and whether the teams are to be added before continuing execution (swapping the job out if the teams are not available) or whenever they become available.

The runtime must decide when it is profitable to get a new team instead of creating more vps on existing teams. It currently bases its decision on the average processor utilitization of one of the teams of the task (extrapolating that utilization to all processors of the whole task). If the utilization is above some threshold, the runtime requests a new team.

The primary mechanism for team retirement is that a vp discovers it is the only vp on its team and that there is no work to do; it then waits for a period of time before relinquishing the team to the operating system and retiring itself.

If it improves performance, we may coalesce small teams.

5. Operating System

The operating system allocates the processing and memory resources of the system among tasks competing for these resources. The memory scheduler first determines which subset of the ready tasks to load into memory. The processor scheduler then determines from the set of in-memory tasks, which tasks to load onto available protection domains. This document deals with the processor scheduler; the memory scheduler is the subject of a companion paper.

There are two types of swapping that occur: swapping between external mass storage and shared memory, and loading and unloading of protection domains in the processors. We say a task is *swapped in* when it has been transferred from external storage to memory, and *swapped out* for the reverse direction. A memory-resident task is *domain-loaded* when it is loaded into protection domains, and *domain-unloaded* when it is unloaded from all protection domains. A task must be swapped in before it can be domain-loaded, and it must be domain-unloaded before it can be swapped out.

Workload

The operating system processor schedulers differentiate between tasks that are highly parallel, often with multiple teams, and those that are not. Highly parallel tasks are likely to also perform I/O in parallel because stopping such a task for I/O seriously degrades performance. Tasks with little or no parallelism can afford to be, and are far more likely to be, willing to block for I/O because the impact of waiting on performance is much less. Parallel tasks tend to use their whole scheduling quantum. Not-very-parallel tasks tend not to, putting more pressure on the scheduler.

Based on our understanding of the exploitable parallelism found in large tasks, we expect that the multi-team tasks in aggregate will impose far greater demand for streams than the single-team (small) tasks. This is in spite of distributed file system server tasks, remote logins, and many other sources of single team tasks containing little parallelism. Each of these single-team tasks will require a tiny fraction of the machine's processing resources for short periods of time.

Because highly parallel tasks have very different processor demand characteristics than minimally parallel tasks, we propose mechanisms for scheduling the processors that differentiate these two classes of workload. Studies of interactive and batch workloads by *Ashok and Zahorjan* support this differentiation.

Processor Resource Management

The protection domains of each processor are divided among single-team and multi-team tasks. A big job processor scheduler (called the PB-scheduler) schedules multi-team tasks, while a small job scheduler (PL-scheduler) schedules single-team tasks. Since multi-team tasks execute on multiple processors, a single PB-scheduler using global knowledge of processor utilization is required per machine. The PB-scheduler *co-schedules* multi-team tasks on the set of protection domains it controls on all processors. Since single-team tasks execute on a single processor, each processor runs its own copy of the PL-scheduler. A multi-team task, or a single-team task with many streams, is assigned to the PB-scheduler; otherwise, the task is assigned to the PL-scheduler with the fewest tasks. The scheduler is chosen when swapping-in completes, providing an opportunity for processor load balancing.

Both schedulers are event driven and similar in design to the *Unix Esched scheduler* and the improved variants due to *Straathof* and *Essick*. These schedulers loop forever processing events from an input queue. The scheduler dequeues an event and creates a kernel privileged daemon stream to examine the event and take appropriate action. The scheduler continues to dequeue events and spawn daemons until the queue is empty. Events are IPC messages sent using the privileged IPC mechanism.

6. Processor Scheduling

Early research on processor scheduling for parallel machines highlighted the need for co-scheduling (also known as *gang scheduling*), *i.e.* ensuring that the parallel activities of a job execute together. Scheduling in this way avoids inefficiencies due to synchronization when some activities are blocked waiting for others that are unscheduled. On traditional multiprocessors, scheduling requires that either a processor be allocated to a single job or that context switching among multiple jobs assigned to a processor be globally coordinated. The former strategy is known as *space-sharing*, in which the processors are partitioned among the jobs in the system. The latter *time-sharing* strategy is generally

realized by round-robin scheduling policies (*see Leutenegger and Vernon*) or distributed hierarchical control (*see Feitelson and Rudolph*) that rotate possession of the processors among the jobs in the system in a coordinated manner.

Studies by *Zahorjan and McCann* and *Setia* comparing the performance of space-sharing and time-sharing systems conclude that space-sharing policies make more efficient use of processors that might otherwise be idle when running a single job. However, in dividing the processors among multiple jobs, space-sharing policies often allocate fewer processors to a job than the job's parallelism may allow. *Tucker and Gupta* describe ''process-control'', an approach that adapts a job's parallelism to the processors available to it. Also, efficient space-sharing policies must be able to adapt the partition sizes and configurations based on changing system load; see *Zahorjan and McCann, McCann, Vaswani, and Zahorjan, McCann and Zahorjan, Naik, Setia, and Squillante, Naik, Setia, and Squillante*, and *Sevcik.*

Tera's multi-threaded architecture makes it possible to combine space-sharing and time-sharing and realize the benefits of both. Multiple tasks execute simultaneously on a single processor. The operating system space-shares the available protection domains among the tasks in the system. A task is allocated as many protection domains as it has teams. Teams from different tasks executing on the same processor compete for hardware streams. The operating system is not involved in this stream allocation or management.

The goals of Tera processor scheduling are to use processing resources efficiently, ensure fairness and starvation avoidance, provide good interactive response time, and supply effective mechanisms for job differentiation.

Achieving high processor utilization requires between 20 and 80 streams. Teams from single-team tasks are likely to have only a few streams, while teams from multi-team tasks will have many more streams (usually 20 to 40). Given 16 protection domains per processor, a processor running only single-team tasks may not have enough streams to keep the processor busy.

To obtain the desired mix of teams from multi-team and single-team tasks, the protection domains on each processor are divided between the multi-team task scheduler (PB-scheduler) and the per-processor single-team task schedulers (PL-schedulers). We expect a processor to be able to service two or three teams from multi-team tasks; call this number the *target-mt-per-proc.*

The PB-scheduler tries to place as many as *target-mt-per-proc* teams from multi-team tasks on every processor, while the per-processor PL-schedulers use the remaining protection domains. The PB-scheduler can acquire additional protection domains from the PL-schedulers. Similarly, when the PB-scheduler has an unneeded protection domain, it can return it to a PL-scheduler. When a task changes from single-team to multi-team status, it migrates from its PL-scheduler to the PB-scheduler. The converse migration does not occur because we expect shrunken tasks will widen again soon.

The processor schedulers do not consider processor utilization. The workload of a team (number of streams) changes so much faster than the operating system can make scheduling decisions that the information cannot be used to predict utilization. In contrast, the runtime can base its load on total processor utilization because it makes workload decisions far more often. We therefore assume that teams from multi-team tasks exert equivalent workloads on the processors. The workload from single-team tasks is also assumed identical.

Single-Team Tasks: the PL-schedulers

Since single-team tasks execute in a single protection domain, their scheduling does not require global knowledge or coordination among processors. Independent instances of the PL-scheduler schedule single-team tasks on a per-processor basis. The PL-schedulers differentiate memory-resident tasks using a priority mechanism similar to the Unix model (see Bach, for example). A newly created task is given a high priority. The priority is decremented if a full quantum is used and incremented when a task unblocks or becomes starved. Site administrators can modify the amount of priority change for each of these events.

As discussed above, a PL-scheduler is assigned to each newly swapped-in single-team task. The binding of a task to a PL-scheduler persists until the task swaps out, exits, or is explicitly sent to another PL-scheduler. The PL-scheduler allocates a protection domain and domain-loads the task unless no protection domains are available; in that case, it places the task on its ready queue. As quanta expire and its tasks block and unblock (or suspend and resume in response to job control), the PL-scheduler moves the tasks among its queues in the conventional manner, always attempting to run as many of its ready tasks of high priority as possible.

Selecting a processor at swap-in time seeks to maintain a balanced load. If in-swapping is infrequent, load imbalance can occur. Based on previous research by *Eager, Lazowska, and Zahorjan* and *Eager et al.*, PL-schedulers migrate work in two situations. When a new task is created (typically by a Unix `fork()` call), or when the ready queue length exceeds a threshold, a PL-scheduler is selected at random and its ready queue is examined. If that queue is empty, tasks are migrated to it. How many tasks migrate depends on how many of the recipient's protection domains are free.

To prevent starvation, the PL-scheduler periodically sweeps its ready queue and increases the priority of tasks that have not run since the last check.

Multi-Team Tasks: the PB-scheduler

Scheduling multi-team tasks presents the problem of allocating protection domains on multiple unique processors in order to co-schedule the task's teams. The PB-scheduler allocates *target-mt-per-proc* protection domains on each processor to the multi-team tasks.

The primary goal of the PB-Scheduler is high processor utilization. The scheduler also ensures that:

1. every processor has at least one and no more than *target-mt-per-proc* teams from multi-team tasks;
2. teams from the same task will be on different processors except when the number of teams is greater than the number of processors; and
3. every multi-team task runs at least once during its memory dwell time (the allotted time the task is memory resident).

Deciding whether to load fewer than *target-mt-per-proc* teams on each processor (goal 1) relates to the user runtime. The user runtime may request a team to exploit increasing parallelism in the task. If the PB-scheduler packs each processor fully, such a request would cause the PB-scheduler to refuse the request or to domain-unload the job until more resources become available. By leaving some protection domains free, the scheduler can service these requests promptly. Domains that are freed as a task's parallelism diminishes can also be used to service growth of other tasks. Determining a correct balance

is difficult, and is a parameter that will be tuned with experience.

To accomplish goal 3, the PB-scheduler uses round-robin scheduling with a processor quantum small enough compared to memory dwell times to ensure every task runs at least once while it is memory resident. Each round is called a cycle.

Team assignment

Assigning teams to protection domains could be done using a standard *first-fit* approach, picking the first protection domain number that can hold the task (Figure 4a). This leads to poor utilization of the protection domains. For example, in Figure 4a if Task 3 needs two protection domains it cannot be scheduled.

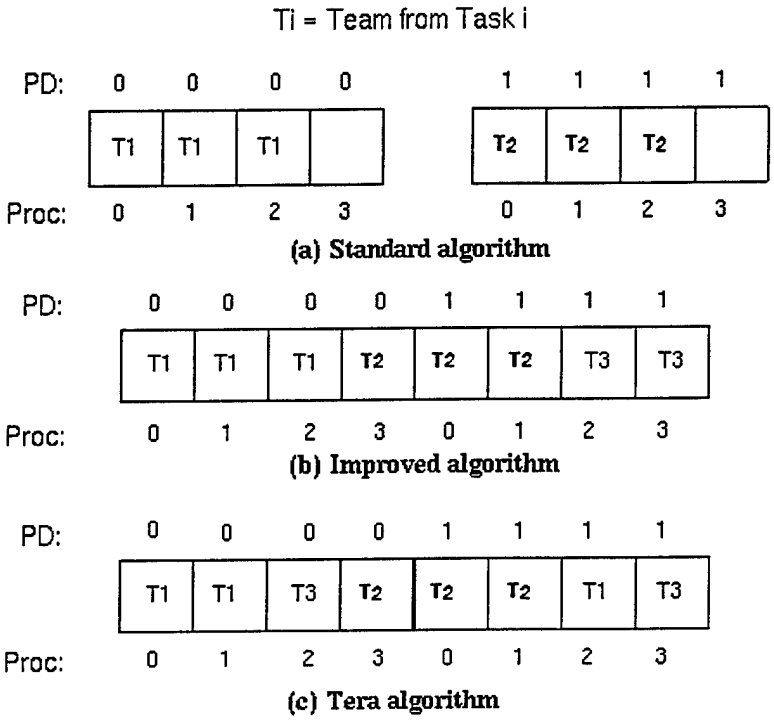

Figure 4. Approaches to Team Assignment

If we order the available (protection domain, processor) pairs lexicographically and assign tasks using first-fit on the ensemble, utilization improves as shown in Figure 4b. We can also handle more teams than processors, a situation that might result from running a checkpoint file on a degraded system. Fragmentation caused by domain-unloads still leads to poor utilization, however; see Figure 5a.

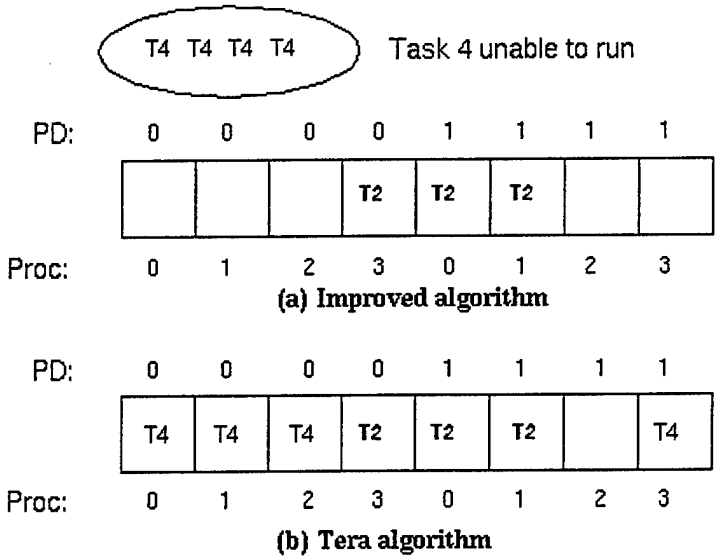

Figure 5. After Tasks 1 and 3 Exit and Task 4 Arrives

A better idea is to use an approach based on bit vectors that iteratively searches for free protection domains on distinct processors. The result will not be lexicographically contiguous in general (Figure 4c). Bit vector operations are particularly efficient on the Tera MTA. The algorithm will succeed in finding distinct processors whenever it is possible to do so; see Figure 5b.

Scheduling a Cycle

At the start of a cycle, all tasks have equal opportunity to be scheduled. Tasks waiting to run reside in the *waiting* queue. The PB-scheduler assigns as many tasks as will fit into the available protection domains and sets an alarm for the processor quantum. All the tasks run for the same amount of time.

When the processor quantum expires, the PB-scheduler conceptually removes the loaded tasks and attempts to schedule tasks in the following order:

1. tasks in the *anti-starvation* queue (described below)
2. tasks that recently arrived and were scheduled in the previous quantum (described below)
3. tasks waiting to run in this cycle (on the *waiting* queue)
4. tasks that have already run in this cycle (on the *ran* queue)
5. tasks currently loaded

After scheduling the next quantum, daemons domain-unload those tasks not scheduled for the next quantum and domain-load the new ones in parallel. domain-unloaded tasks are placed in the ran queue. When the waiting queue is empty, the cycle is complete and the ran queue becomes the waiting queue.

The PB-scheduler tries to assign protection domains to newly arriving tasks regardless of how much time remains in this processor quantum. Since the new task will, on average, only receive a half of a quantum, the PB-scheduler places the task in a list to be scheduled before those on the waiting queue.

To avoid starvation, the PB-scheduler periodically checks tasks in the waiting queue for starvation. Any tasks that have not received service since the last check are placed in an *anti-starvation* queue. This queue is first in line for scheduling in the next quantum.

7. Summary

Scheduling of execution resources appears at all levels in a Tera System: in compiler generated code, in the runtime system, and in the operating system. The overall goal of our design efforts has been to ensure that these three major scheduling systems work effectively and efficiently both within their respective domains and in their interactions with one another.

The compiler schedules instructions. It also acquires and schedules stream resources for the implicit parallelism of a program. The overheads involved in scheduling done by the compiler are typically negligible, and the time frame of a compiler's schedule is normally shortest of all.

The runtime acquires and schedules stream resources for executing explicit program parallelism. It also imposes an order on the execution of parallel work. There is slightly more overhead associated with this kind of parallel work than that scheduled by the compiler. Appropriately, the granularity of the work is expected to be somewhat larger, and the scheduling time frame, too, is greater than that considered by the compiler.

The operating system schedules protection domain resources among jobs contending for the system. The time frame considered by the processor scheduler is much greater than that of the runtime; and, again, this is natural since each piece scheduled may consist of many futures or compiler scheduled parallel loops.

Using multiple schedulers, each specifically designed for a particular level in a hierarchy of time frames and work granularities, is a natural approach for scheduling parallel work on a multiprogrammed machine. We expect our designs to be particularly efficient, however, because they are able to execute almost independently of each other. Interactions between the operating system and the runtime systems are limited to occasional requests for additional protection domains. The runtime handles traps, creates additional streams, and estimates workload without operating system help. Compiler generated code, once assigned by the runtime to specific teams, operates independently of the runtime, reserving, creating and destroying streams immediately, via hardware instructions. We believe the decomposition of scheduling obligations in this way will lead to negligible overheads and very high processor utilizations.

The processor schedulers use efficient mechanisms for bit vector iteration provided by the Tera MTA to efficiently co-schedule jobs onto protections domains. The runtime growth policy is based on relatively recent competitive-style algorithms, and is extremely simple while still promising robust adherence to predictable performance bounds.

Our schedulers appear reasonable in simulation, though due to the relatively slow speed of simulation, we can run only small workloads. We expect to refine our algorithms with experience on the real machine, and we eagerly await the arrival of the Tera MTA prototype.

8. References

R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith.
The Tera computer system.
In *1990 International Conference on Supercomputing*, June 1990.

I. Ashok and J. Zahorjan.
Scheduling a mixed interactive and batch workload on a parallel, shared memory supercomputer.
In *Supercomputing 1992*, Nov 1992.

M. J. Bach.
The Design of the Unix Operating Systems.
Prentice-Hall, Inc., 1986.

D.L. Eager, E.D. Lazowska, and J. Zahorjan.
Adaptive load sharing in homogeneous distributed systems.
IEEE Transactions on Software Engineering, May 1986.

D.L. Eager, E.D. Lazowska, and J. Zahorjan.
A comparison of receiver-initiated and sender-initiated adaptive load sharing.
Performance Evaluation, March 1986.

R. Essick.
An event-based fair share scheduler.
In *Winter 90 USENIX Conference*, January 1990.

D. G. Feitelson and L. Rudolph.
Distributed hierarchical control for parallel processing.
Computer, 23(5), May 1990.

D. G. Feitelson and L. Rudolph.
Gang scheduling performance benefits for fine-grain synchronization.
Journal of Parallel and Distributed Computing, 16(4), December 1992.

A. Gottleib, B. Lubachevsky, and L. Rudolph.
Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors.
ACM Transactions on Programming Languages and Systems, 5(2), April 1983.

A. Gupta, A. Tucker, and S. Urushibara.
The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications.
In *ACM SIGMETRICS Conference*, May 1991.

S. Leutenegger and M. Vernon.
The performance of multiprogrammed multiprocessor scheduling policies.
In *ACM SIGMETRICS Conference*, May 1990.

C. McCann, R. Vaswani, and J. Zahorjan.
A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors.

ACM Transactions on Computer Systems, May 1993.

C. McCann and J. Zahorjan.
Processor allocation policies for message-passing parallel computers.
In *ACM SIGMETRICS Conference*, May 1994.

V. Naik, S. Setia, and M. Squillante.
Performance analysis of job scheduling policies in parallel supercomputing environments.
In *Supercomputing 1993*, November 1993.

V. Naik, S. Setia, and M. Squillante.
Scheduling of large scientific applications on distributed memory multiprocessor systems.
In *6th SIAM Conference on Parallel Processing for Scientific Computation*, March 1993.

J. Ousterhout.
Scheduling techniques for concurrent systems.
In *3rd International Conference on Distributed Computing*, Oct 1982.

S. Setia, M. Squillante, and S. Tripathi.
Processor scheduling on multiprogrammed, distributed memory parallel systems.
In *ACM SIGMETRICS Conference*, May 1993.

K. Sevcik.
Characterization of parallelism in applications and their use in scheduling.
In *ACM SIGMETRICS Conference*, May 1989.

J. H. Straathof, A. A. Thareja, and A. K. Agrawala.
Unix scheduling for large systems.
In *Denver USENIX Conference*, January 1986.

J. H. Straathof, A. A. Thareja, and A. K. Agrawala.
Methodology and results of performance measurements for a new unix scheduler.
In *Washington USENIX Conference*, January 1987.

A. Tucker and A. Gupta.
Process control and scheduling issues for multiprogrammed shared-memory multiprocessors.
In *12th ACM Symposium on Operating System Principles*, December 1989.

D. Wagner and B. Calder.
Leapfrogging: A portable technique for implementing efficient futures.
In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

J. Zahorjan, E. Lazowska, and D. Eager.
Spinning versus blocking in parallel systems with uncertainty.
In *International Sympossium on Performance of Distributed and Parallel Systems*, December 1988.

J. Zahorjan, E. Lazowska, and D. Eager.
The effects of scheduling discipline on spin overhead in shared memory parallel systems.